2015

Efficient modular exponentiation based on multiple multiplications by a common operand

Christophe Negre
DALI (UPVD) and LIRMM (UM2, CNRS) , France

Thomas Plantard
University of Wollongong, thomaspl@uow.edu.au

Jean-Marc Robert
DALI (UPVD) and LIRMM (UM2, CNRS) , France

Publication Details

Negre, C., Plantard, T. & Robert, J. (2015). Efficient modular exponentiation based on multiple multiplications by a common operand. In J. Muller, A. Tisserand & J. Villalba (Eds.), Proceedings of the 2015 IEEE Symposium on Computer Arithmetic (ARITH 22) (pp. 144-151). Piscataway, New Jersey, United States: IEEE.

Efficient modular exponentiation based on multiple multiplications by a common operand

Abstract
The main operation in RSA encryption/decryption is the modular exponentiation, which involves a long sequence of modular squarings and multiplications. In this paper, we propose to improve modular multiplications AB, AC which have a common operand. To reach this goal we modify the Montgomery modular multiplication in order to share common computations in AB and AC. We extend this idea to reduce the cost of multiple modular multiplications AB1,...,ABl by the same operand A. We then take advantage of these improvements in the Montgomery-ladder and SPA resistant m-ary exponentiation algorithms. The complexity analysis shows that for an RSA modulus of size 2048 bits, the proposed improvements reduce the number of word operations (ADD and MUL) by 14% for the Montgomery-ladder and by 5%-8% for the m-ary exponentiations. Our implementations show a speed-up by 8%-14% for the Montgomery-ladder and by 1%-8% for the m-ary exponentiations for modulus of size 1024, 2048 and 4048 bits.

Disciplines
Engineering | Science and Technology Studies

Publication Details
Negre, C., Plantard, T. & Robert, J. (2015). Efficient modular exponentiation based on multiple multiplications by a common operand. In J. Muller, A. Tisserand & J. Villalba (Eds.), Proceedings of the 2015 IEEE Symposium on Computer Arithmetic (ARITH 22) (pp. 144-151). Piscataway, New Jersey, United States: IEEE.

Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand

Christophe Negre*, Thomas Plantard†, Jean-Marc Robert*
* *Team DALI (UPVD) and LIRMM (UM2, CNRS) , France*
† *CCISR, SCIT, (University of Wollongong), Australia*

***Abstract*—The main operation in RSA encryption/decryption is the modular exponentiation, which involves a long sequence of modular squarings and multiplications. In this paper, we propose to improve modular multiplications AB, AC which have a common operand. To reach this goal we modify the Montgomery modular multiplication in order to share common computations in AB and AC. We extend this idea to reduce the cost of multiple modular multiplications $AB_1, \ldots, AB_\ell$ by the same operand A. We then take advantage of these improvements in the Montgomery-ladder and SPA resistant m-ary exponentiation algorithms. The complexity analysis shows that for an RSA modulus of size 2048 bits, the proposed improvements reduce the number of word operations (ADD and MUL) by 14% for the Montgomery-ladder and by 5%-8% for the m-ary exponentiations. Our implementations show a speed-up by 8%-14% for the Montgomery-ladder and by 1%-8% for the m-ary exponentiations for modulus of size 1024, 2048 and 4048 bits.**

***Keywords*-Montgomery multiplication, word level, common operand, modular exponentiation.**

I. Introduction

RSA [7] is the most widely used public key cryptosystem. It is for example used to secure credit card transaction and to generate SSL/TLS certificates with an RSA signature. The main operation in RSA protocols is the modular exponentiation $G^e \mod N$. In order to ensure a sufficient security level, the modulus N is typically of size 1000-4000 bits. The most widely used method to compute a modular exponentiation is the square-and-multiply approach, which consists of a long sequence of a few thousands of squarings and multiplications modulo N.

In practice, such exponentiation methods can be threatened by side channel analysis. For example, by measuring several computation times of a modular exponentiation and exploiting the difference of squaring and multiplication times, an attacker can recover part of the key (cf. [3]). An attacker can also monitor the power consumption and then read the sequence of operations on the power trace and recover the key bits: this is the Simple Power Analysis (SPA) [4]. In this paper, we focus on the variants of the square-and-multiply robust against timing and SPA attacks, by ensuring a regularity of the sequence of multiplications and squarings. Specifically, the considered methods are the Montgomery-ladder [2] and the regular m-ary exponentiations [1].

Let us assume that N is stored in n computer words. A multiplication modulo N consists of a multi-precision integer multiplication followed by a reduction modulo N. The modulus N used in RSA cryptosystem has a random form. Consequently, the reduction modulo N is generally performed using the Montgomery approach [6]. This reduction is thus quite costly since an integer multiplication requires $\cong 3n^2$ word operations (word additions or multiplications) while the reduction also requires $\cong 3n^2$ word operations.

In this paper, we present a modified version of the Montgomery multiplication which performs two multiplications AB, AC with a common operand A. We reduce the complexity by sharing part of the computations involved in the costly reductions modulo N in AB and AC. This reduces the overall complexity of AB, AC by 25%. We extend this idea to improve multiple multiplications $AB_1, \ldots, AB_\ell$ by a common operand A: some redundant operations can be avoided by storing some data in memory and by subsequently reusing them in the multiplications $AB_i, i = 1, \ldots, \ell$. For sufficiently large ℓ, this saves $\cong 50\%$ of word operations per multiplication.

The remaining of the paper is organized as follows. Section II is a review the word level version of the Montgomery modular multiplication. In Section III, we present an algorithm which performs two Montgomery multiplications with a common operand AB, AC, and we extend this approach to reduce the cost of multiple multiplications by a common operand A. In Section IV, a modified versions of the Montgomery-ladder and m-ary exponentiations are presented, which perform part of the modular multiplications using the proposed improvement on multiplications with a common operand. Section V gives a comparison of the complexity of the proposed approaches with their conventional counterparts and provide implementation results. In Section VI, we give a few concluding remarks.

II. Review of Montgomery modular multiplication

We consider an RSA modulus N such that $N < 2^{wn}$. Let A and B be two integers in $[0, N]$. The classical way to compute $Y = A \cdot B \mod N$ is to perform an Euclidean

division of $C = A \times B$ by N

$$Q \leftarrow \lfloor (A \times B)/N \rfloor \text{ and } Y \leftarrow A \times B - QN,$$

and Y satisfies $Y < N$. In other words, the most significant bits of C are cleared by subtracting QN. Montgomery in [6] performs this reduction in a different way: instead of clearing the most significant bits of $C = A \times B$, one clears the least significant bits, by first computing

$$Q \leftarrow A \times B \cdot (-N^{-1}) \mod 2^{wn},$$

and then computing the following exact division

$$Y \leftarrow (X + Q \cdot N)/2^{wn}.$$

This produces Y which satisfies $Y = X \times 2^{-wn} \mod N$ and $Y < 2N$.

When one needs to compute a long sequence of modular multiplications, the elements modulo N are generally set in the so-called Montgomery representation $\widetilde{A} = A \cdot 2^{wn} \mod N$. Indeed, a Montgomery multiplication computes

$$\widetilde{A} \cdot \widetilde{B} \cdot 2^{-wn} \mod N = (AB) \cdot 2^{wn} \mod N$$

which is the Montgomery representation of the product.

We now review the word level variant of the Montgomery representation which can be found in [5]. We first review the two basic operations involved in this algorithm which are the word-multiplication and the small reduction.

Word-multiplication: We first study the operation $a \cdot B$ where a is a w-bit integer and $B = (b_{n-1}, \ldots, b_0)_{2^w}$ is an nw-bit integer. This operation will be the building block of most of the algorithms presented in this paper. The basic method to compute $a \cdot B$ consists in expanding the product relatively to $B = \sum_{j=0}^{n-1} b_j 2^{wj}$:

$$a \cdot B = \sum_{j=0}^{n-1} a b_j 2^{wj}.$$

Then, the products ab_j for $j = 0, \ldots, n-1$ are sequentially computed and added to the intermediate result X as shown Algorithm 1.

Algorithm 1 Word-multiplication

Require: a, B with $B = (b_{n-1}, \ldots, b_0)_{2^w}$ and $0 \le a, b_i < 2^w$
Ensure: $X = (x_n, \ldots, x_0)_{2^w}$ with $X = a \cdot B < 2^{(n+1)w}$
  $\{v, u\} \leftarrow a \cdot b_0$ //u is the lower and v the upper part
  $x_0 \leftarrow u$, $x_1 \leftarrow v$
  for $j = 1$ **to** $n - 1$ **do**
    $\{v, u\} \leftarrow a \cdot b_j$ //u is the lower and v the upper part
    $x_i \leftarrow \text{ADDC}(x_i, u)$ // addition with carry
    $x_{i+1} \leftarrow v$
  $x_n \leftarrow \text{ADDC}(x_n, 0)$ // absorption of the last carry
  return $X = (x_n, \ldots, x_0)_{2^w}$

The above Word-multiplication algorithm requires n word multiplications (MUL) and n word additions (ADD).

Small Montgomery reduction: The small reduction is a version of the Montgomery modular reduction which reduces an integer A modulo N by w bits. This method is depicted in Algorithm 2.

Algorithm 2 SmallRed

Require: A modulus $N < 2^{wn-2}$ and a positive integer $X = (x_{n'-1}, \ldots, x_0)_{2^w}$ of n' words and $N' = (-N^{-1}) \mod 2^w$
Ensure: $Y = X \cdot 2^{-w} \mod N$ with $Y < X/2^w + N$
1: $q \leftarrow x_0 \cdot N' \mod 2^w$
2: $Y \leftarrow (X + q \cdot N)/2^w$
3: **return** Y

The complexity of SmallRed with an n' word input X is $\max(n', n+1) + n$ ADD and $n+1$ MUL. The following lemma establishes some basic facts concerning the SmallRed algorithm which will be useful in the sequel.

Lemma 1. *Let $X \ge 0$ be the input of Algorithm 2. Then, the output Y satisfies*

i) $Y < X/2^w + N$ and $Y = X2^{-w} \mod N$.
ii) If $X < 2N$ then $Y < 2N$.

Proof:

i) By construction, one has $(X + q \cdot N) = 0 \mod 2^w$. Thus the division by 2^w (Step 2 of Algorithm 2) is exact and Y is well defined. This implies that $Y2^w = (X + q \cdot N) \equiv X \mod N$ and then $Y \equiv X \cdot 2^{-w} \mod N$. It is also straightforward to get $Y = (X + qN)/2^w < X/2^w + N$ since $q < 2^w$.
ii) In the special case $X < 2N$, one has $Y = (X + qN)/2^w < 2N/2^w + N < 2N$.

■

Word level Montgomery multiplication: We review the word level form of a Montgomery multiplication. Let us consider two integers A and B of size $< N$. The idea is to interleave Word-multiplications (Algo. 1) with SmallReds to maintain the intermediate product Y in the range $[0, 2N]$ at the end of each loop turn. A final subtraction by N produces $Y < N$. This method is detailed in Algorithm 3.

The complexity of Algorithm 3 is evaluated step by step in Table I by using the complexity of Word-multiplication.

A word level Montgomery squaring can be computed more efficiently than a Montgomery multiplication. Indeed, some of the products $a_i a_j$ are redundant and thus can be avoided:

$$\begin{aligned} A^2 &= \sum_{i=0}^{n-1} a_i 2^{w(2i)} (a_i + 2 \sum_{j=1}^{n-i-1} a_{i+j} 2^{wj}) \\ &= \sum_{i=0}^{n-1} a_i 2^{w(2i)} \widetilde{A}_i. \end{aligned} \quad (1)$$

The integers $\widetilde{A}_i = (a_i + 2 \sum_{j=1}^{n-i-1} a_{i+j} 2^{wj})$ can be deduced

Algorithm 3 Word level Montgomery multiplication [5]

Require: $N < 2^{wn-2}$ the modulus, w the word size, $A = (a_{n-1}, \ldots, a_0)_{2^w}$ and $B = (b_{n-1}, \ldots, b_0)_{2^w}$ two n-word integers in $[0, N]$ and $N' = (-N^{-1}) \mod 2^w$
Ensure: $Y = A \cdot B \cdot 2^{-wn} \mod N$

$Y \leftarrow a_0 \cdot B$
$q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$
$Y \leftarrow (Y + q \cdot N)/2^w$
for $i = 1$ **to** $n - 1$ **do**
  $Y \leftarrow Y + a_i \cdot B$
  $q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$
  $Y \leftarrow (Y + q \cdot N)/2^w$
if $Y > N$ **then**
  $Y \leftarrow Y - N$
return Y

Table I
COMPLEXITY OF ALGORITHM 3

	Operations	# ADD	# MUL
Step 1	$a_0 \times B$	n	n
Step 2	$\|Y\|_{2^w} \cdot N'$	0	1
Step 3	$q \times N$ $Y + (qN)$	n $n+1$	n 0
$n-1$ Step 5	$a_i \times B$ $Y + (a_i B)$	$(n-1)n$ $(n-1)(n+1)$	$(n-1)n$ 0
$n-1$ Step 6	$\|Y\|_{2^w} \cdot N'$	0	$n-1$
$n-1$ Step 7	$q \times N$ $Y + (qN)$	$(n-1)n$ $(n-1)(n+1)$	$(n-1)n$ 0
Step 9	$Y - N$	n	0
Total		$4n^2 + 2n - 1$	$n(2n+1)$

from $A' = 2A = (a'_{n-1}, \ldots, a'_0)_{2^w}$ as

$$\widetilde{A}_i = (a'_{n-1}, \ldots, a'_{i+2}, |2a_{i+1}|_{2^w}, a_i)_{2^w}.$$

With the formulation (1), we derive the word level Montgomery squaring shown in Algorithm 4.

The overall complexity of Algorithm 4 is obtained by adding the contribution of each step: thus, one has $3n^2 + 5n - 1$ ADD and $\frac{3n^2}{2} + \frac{5n}{2} - 1$ MUL.

Table II gathers all the complexities of the algorithms reviewed in this section: SmallRed, Montgomery multiplication (Algorithm 3) and Montgomery squaring (Algorithm 4).

Table II
COMPLEXITIES FOR MONTGOMERY MULTIPLICATION AND SQUARING

Operation	# ADD	# MUL
SmallRed (Algo. 2) with an n'-word input	$\max(n', n+1) + n$	$n+1$
MontSqu (Algo. 4)	$3n^2 + 5n - 1$	$\frac{3n^2}{2} + \frac{5n}{2} - 1$
MontMul (Algo. 3)	$4n^2 + 2n - 1$	$2n^2 + n$

Algorithm 4 Word level Montgomery Squaring

Require: A, with $A = (a_{n-1}, \ldots, a_0)_{2^w}$ with $0 \le a_i < 2^w$ where w is the word size, $N' = -N^{-1} \mod 2^w$
Ensure: $Y \equiv A^2 \times 2^{-wn} \mod N$ and $X < N$

$A' \leftarrow A + A$ // n ADD
$\widetilde{A}_0 \leftarrow (a'_{n-1}, \ldots, a'_2, |2a_1|_{2^w}, a_0)_{2^w}$ // 1 ADD
$Y \leftarrow \widetilde{A}_0 \cdot a_0$ // n ADD and n MUL
$q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$ // 1 MUL
$Y \leftarrow (Y + q \cdot N)/2^w$ // $2n+1$ ADD and n MUL
for $i = 1$ **to** $(n-1)$ **do**
  $\widetilde{A}_i \leftarrow (a'_{n-1}, \ldots, a'_{i+2}, |2a_{i+1}|_{2^w}, a_i)_{2^w}$ //$n-1$ ADD
  $Y \leftarrow Y + \widetilde{A}_i \cdot a_i \cdot 2^{wi}$ // $n^2 - 1$ ADD
  // and $\frac{(n-1)n}{2}$ MUL
  $q \leftarrow |Y|_{2^w} \cdot N' \mod 2^w$ // $n-1$ MUL
  $Y \leftarrow (Y + q \cdot N)/2^w$ // $(n-1)(2n+1)$ ADD
  // and $(n-1)n$ MUL
if $Y > N$ **then**
  $Y \leftarrow Y - N$ // n ADD
return Y

III. IMPROVED MONTGOMERY MULTIPLICATIONS WITH A COMMON OPERAND

In this section, we present some improvements of multiple Montgomery multiplications with a common operand. We first deal with the case $A \cdot B, A \cdot C$ and then generalize this result to $A \cdot B_i, i = 1, \ldots, \ell$.

A. Improved combined multiplications $A \cdot B, A \cdot C$

We present in this section an algorithm which takes as input A, B, C and N, and outputs $ABR^{-1} \mod N$ and $ACR^{-1} \mod N$ with $R = 2^{w(n+1)}$. Our goal is to share some common computations performed in the Montgomery products $ABR^{-1} \mod N$ and $A\,CR^{-1} \mod N$. To reach this goal, the product ABR^{-1} relatively to B are expanded as follows:

$$\begin{aligned} ABR^{-1} &= (\sum_{j=0}^{n-1} b_j 2^{wj}) \cdot A \cdot 2^{-w(n+1)} \mod N \\ &= \sum_{j=0}^{n-1} b_j A \cdot 2^{-w(n+1-j)} \mod N \\ &= \sum_{j=0}^{n-1} b_j (A \cdot 2^{-w(n-1-j)} \mod N) 2^{-2w} \mod N \\ &= (\sum_{j=0}^{n-1} b_j A^{(j)}) \cdot 2^{-2w} \mod N \end{aligned} \quad (2)$$

where $A^{(j)} = A 2^{-w(n-1-j)} \mod N$ for $j = 0, \ldots, n-1$. With the same for $A \cdot C \cdot R^{-1}$, one obtains

$$A \cdot C \cdot R^{-1} = (\sum_{j=0}^{n-1} c_j A^{(j)}) \cdot 2^{-2w} \mod N. \quad (3)$$

We notice that the expression in (2) for ABR^{-1} and the expression in (3) for ACR^{-1} contain the terms $A^{(j)} = A2^{-w(n-1-j)} \mod N, j = 0, 1, \ldots, n-1$. To compute $A^{(j)}$, we start from $A^{(n-1)} = A$ and we apply a sequence of $n-1$ SmallReds as follows:

$$\begin{aligned}
\text{SmallRed}(A^{(n-1)}) &= A \cdot 2^{-w} \mod N \\
&= A^{(n-2)}, \\
\text{SmallRed}(A^{(n-2)}) &= (A \cdot 2^{-w}) \cdot 2^{-w} \mod N \\
&= A^{(n-3)}, \\
&\vdots \qquad\qquad \vdots \\
\text{SmallRed}(A^{(1)}) &= (A \cdot 2^{-w(n-2)}) \cdot 2^{-w} \mod N \\
&= A \cdot 2^{-w(n-1)} = A^{(0)}.
\end{aligned}$$

Therefore, one computes AB and AC as follows: the terms $A^{(j)}$ are computed and then $b_j A^{(j)}$ and $c_j A^{(j)}$ are accumulated in Y and Z for $j = 0, \ldots, n-1$. From Lemma 1, if $A < 2N$, then all the values $A^{(j)}, j = 0, \ldots, n-1$, satisfy $A^{(j)} < 2N$. The resulting accumulation $\sum_{j=0}^{n-1} b_j A^{(j)} < 2Nn2^w$ in Y is reduced to a value $< 2N$ by performing two consecutive SmallReds. At the same time, these two SmallReds produce the missing factor 2^{-2w} in (2). The same is done for Z to get (3). This approach is depicted in Algorithm 5. Its complexity is evaluated step by step in Table III.

Algorithm 5 CombinedMontMul(A, B, C)

Require: the modulus $N < 2^{wn-2}$, three integers $A = (a_{n-1}, \ldots a_0)_2, B = (b_{n-1}, \ldots b_0)_2, C = (c_{n-1}, \ldots c_0)_2$ such that $A, B, C < 2N$, w the word size, $R = 2^{w(n+1)}$ the Montgomery constant.
Ensure: $Y = A \cdot B \cdot R^{-1} \mod N$ and $Z = A \cdot C \cdot R^{-1} \mod N$
1: $X \leftarrow A$
2: $Y \leftarrow b_{n-1} \cdot X,\ Z \leftarrow c_{n-1} \cdot X$
3: **for** $j = n-2$ **downto** 0 **do**
4: $\quad q \leftarrow |X|_{2^w} N' \mod 2^w$
5: $\quad X \leftarrow (X + q \cdot N)/2^w \quad // = A^{(j)}$ for $j = n-2, \ldots, 0$
6: $\quad Y \leftarrow Y + b_j \cdot X,\ Z \leftarrow Z + c_j \cdot X$
7: $Y \leftarrow$ SmallRed$(Y),\ Z \leftarrow$ SmallRed(Z)
8: $Y \leftarrow$ SmallRed$(Y),\ Z \leftarrow$ SmallRed(Z)
9: **return** Y and Z

B. Multiple multiplications with a common operand

In this subsection, we extend the idea used in the CombinedMontMul algorithm to multiple multiplications by a common operand. Specifically, given a fixed element $A \in \{0, \ldots, N-1\}$ and a sequence of $B_i, i = 1, \ldots, \ell$, we want to compute:

$$\begin{aligned}
Y_1 &= A \cdot B_1 \cdot 2^{-w(n+1)} \mod N, \\
Y_2 &= A \cdot B_2 \cdot 2^{-w(n+1)} \mod N, \\
&\vdots \\
Y_\ell &= A \cdot B_\ell \cdot 2^{-w(n+1)} \mod N.
\end{aligned}$$

As in Subsection III-A, we expand each multiplication $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ relatively to B_i and rewrite the product as follows:

$$\begin{aligned}
A \cdot B_i \cdot 2^{-w(n+1)} &= (\sum_{j=0}^{n-1} b_{i,j} 2^{wj}) \cdot A^{-w(n+1)} \mod N \\
&= (\sum_{j=0}^{n-1} b_{i,j} A^{(j)}) \cdot 2^{-2w} \mod N. \quad (4)
\end{aligned}$$

One may notice that the above expression of $A \cdot B_i \cdot 2^{-w(n+1)}$ contains $A^{(j)} = A \cdot 2^{-w(n-1-j)} \mod N$ for $j = 0, \ldots, n-1$. We propose to precompute these n terms $A^{(j)}$ and store them in memory. They are computed through a sequence of $n-1$ SmallReds as shown Algorithm 6.

Table III
COMPLEXITY OF ALGORITHM 5

	Operations	# ADD	# MUL
Step 2	$b_{n-1} \cdot X$	n	n
Step 2	$c_{n-1} \cdot X$	n	n
$(n-1)$ Step 4	$\lvert X \rvert_{2^w} \cdot N'$	0	$n-1$
$(n-1)$ Step 5	$q \cdot N$ $X + (qN)$	$(n-1)n$ $(n-1)(n+1)$	$(n-1)n$ 0
$(n-1)$ Step 6	$b_j \cdot X$ $Y + (b_j X)$	$(n-1)n$ $(n-1)(n+2)$	$(n-1)n$ 0
$(n-1)$ Step 6	$c_j \cdot X$ $Z + (c_j X)$	$(n-1)n$ $(n-1)(n+2)$	$(n-1)n$ 0
Step 7	$2\times$ SmallRed with $n' = n+2$	$4n+4$	$2n+2$
Step 8	$2\times$ SmallRed with $n' = n+1$	$4n+2$	$2n+2$
Total		$6n^2+9n+1$	$3n^2+4n+3$

Algorithm 6 PrecompMultByComOp(A)

Require: $A, N \in \mathbb{Z}$ with $A < 2N$ and $N < 2^{nw-2}$, a precomputed value $N' = (-N^{-1}) \mod 2^w$.
Ensure: $A^{(0)}, \ldots, A^{(n-1)}$ such as $A^{(j)} = 2^{-w(n-1-j)} \cdot A \mod N$ and $A^{(j)} < 2N$
1: $A^{(n-1)} \leftarrow A \mod N$
2: **for** $j = n-2$ **downto** 0 **do**
3: $\quad A^{(j)} \leftarrow$ SmallRed$(A^{(j+1)})$
4: **return** $A^{(0)}, \ldots, A^{(n-1)}$

Complexity of Algorithm 6. This algorithm consists of $n-1$ SmallReds, all with an n-word input. Using the complexity of SmallRed shown in Table II, we obtain the complexity of Algorithm 6, which is $(2n+1)(n-1)$ ADD and (n^2-1) MUL.

Now, with the n precomputed terms $A^{(0)}, \ldots, A^{(n-1)}$, one can compute any of the products $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ for $i = 1, \ldots, \ell$ using (4). Indeed, one computes $Y = \sum_{j=0}^{n-1} b_{i,j} A^{(j)}$ through the sequence $Y \leftarrow Y + b_{i,j} A^{(j)}$ for $j = 0, 1, \ldots, n-1$, and Y satisfies $Y = A \cdot B_i \cdot 2^{-w(n-1)}$ and $Y < 2Nn2^w$. One needs two final SmallReds in order to reduce Y to a value $< 2N$ and also to multiply Y by 2^{-2w} resulting in $Y = AB_i 2^{-w(n+1)} \mod N$, as required. In Algorithm 7, this strategy is used to compute $A \cdot B \cdot 2^{-(w(n+1)} \mod N$ for an arbitrary input B in $[0, 2N]$.

Algorithm 7 MultByComOp$(B, A^{(0)}, \ldots, A^{(n-1)})$

Require: the modulus $N < 2^{wn-2}$, an integer $B = (b_{n-1}, \ldots b_0)_{2^w}$ such that $B < 2N$ and $A^{(j)} = 2^{-w(n-1-j)} \cdot A \mod N$ for $j = 0, \ldots, n-1$
Ensure: $Y = A \cdot B \cdot 2^{-w(n+1)} \mod N$ with $Y < 2N$
1: $Y \leftarrow b_0 \cdot A^{(0)}$
2: **for** $j = 1$ **to** $n-1$ **do**
3: $\quad Y \leftarrow Y + b_j \cdot A^{(j)}$
4: $Y \leftarrow$ SmallRed(Y)
5: $Y \leftarrow$ SmallRed(Y)
6: **return** Y

The complexity of Algorithm 7 is evaluated step by step in Table IV.

Table IV
COMPLEXITY OF ALGORITHM 7

	Operations	# ADD	# MUL
Step 1	$b_0 \cdot A^{(0)}$	n	n
Step 3 $\times(n-1)$	$b_j \cdot A^{(j)}$ $Y + (b_j A^{(j)})$	$(n-1)n$ $(n-1)(n+2)$	$(n-1)n$ 0
Step 5	SmallRed with $n' = n+2$	$2n+2$	$n+1$
Step 6	SmallRed with $n' = n+1$	$2n+1$	$n+1$
Total		$2n^2+5n+1$	n^2+2n+2

The computation of ℓ multiplications $A \cdot B_i \cdot 2^{-w(n+1)} \mod N$ for $i = 1, \ldots, \ell$ consists of one execution of Algorithm 6 and ℓ executions of Algorithm 7. The cost for ℓ multiplications using this strategy is then $\ell(2n^2+5n+1) + (2n+1)(n-1)$ ADD and $\ell(n^2+2n+2) + (n^2-1)$ MUL.

C. Complexity comparison

In Table V, we report the complexity of the proposed CombinedMontMul and MultByComOp approaches. We also provide the complexities of the usual approaches which perform two independent MontMuls for AB, AC and one MontMul and one MontSqu for AB, A^2. For ℓ multiplications by a common operand, we provide the complexity of ℓ MontMuls.

Table V
COMPLEXITY COMPARISON FOR AB, AC AND AB, A^2 AND MULTIPLE MULTIPLICATIONS BY A

Operation	Algorithm	# ADD	# MUL
AB, AC	Two MontMuls	$8n^2+4n-2$	$4n^2+2n$
AB, A^2	MontMul and MontSqu	$7n^2+7n-2$	$\frac{7}{2}n^2+\frac{7}{2}n-1$
AB, AC	CombinedMontMul	$6n^2+9n+1$	$3n^2+4n+3$
AB_i, $i = 1, \ldots, \ell$	ℓ MontMuls	$\ell(4n^2+2n-1)$	$\ell(2n^2+n)$
AB_i, $i = 1, \ldots, \ell$	$\ell\times$ MultByComOp and one PrecomMultByComOp	$\ell(2n^2+5n+1)$ $+(2n+1)(n-1)$	$\ell(n^2+2n+2)$ $+(n^2-1)$

One can check that our proposed CombinedMontMul for AB, AC is better by 25% than the classical approach of two independent MontMuls. For AB, A^2, the CombinedMontMul approach reduces the complexity by 13% compared to MontMul and MontSqu.

For ℓ multiplications $AB_1, \ldots, AB_\ell$, we notice that when $\ell \geq 2$, the proposed approach is more efficient by 25% compared to ℓ independent MontMuls. When $\ell \geq n$ and when n is large enough, the complexity of our approach tends to be 50% less than the one of ℓ independent MontMuls.

IV. EXPONENTIATION WITH IMPROVED MULTIPLICATIONS BY A COMMON OPERAND

In this section, we consider modular exponentiation algorithms which are robust against SPA and timing attacks: Montgomery-ladder [2] and regular m-ary exponentiation [1]. We use the approaches presented in the previous section to improve these exponentiations modulo N. Specifically, we show that the Montgomery-ladder and the right-to-left m-ary exponentiation can take advantage of the CombinedMontMul algorithm and that the left-to-right exponentiation can take advantage of the MultByComOp algorithm.

A. Montgomery-ladder with CombinedMontMul

The Montgomery-ladder (cf. [2]) computes an exponentiation $G^e \mod N$ through a sequence of operations of the form $A \cdot B, A \cdot A$. These operations can be performed using the CombinedMontMul algorithm since CombinedMontMul(A, B, A) returns $A \cdot B$ and $A \cdot A$. Based on the complexities reported in Table V, the use of CombinedMontMul is more efficient than the classical approach based on MontMul and MontSqu. This will reduce the complexity of the Montgomery-ladder. The resulting modified Montgomery-ladder is given Algorithm 8.

Complexity of Algorithm 8. The operations performed in Algorithm 8 consist of two Montgomery multiplications, two SmallReds and k CombinedMontMul (in the **for** loop).

Algorithm 8 Montgomery-ladder with CombinedMontMul

Require: $N < 2^{wn-1}$ and $G \in \{0, \ldots, N-1\}$ and an exponent $e = (e_{k-1}, \ldots e_0)_2$, w the word size and $R = 2^{w(n+1)}$ the Montgomery constant.
Ensure: $G^e \mod N$
1: $X_0 \leftarrow R \mod N$
2: //conversion $X_1 \leftarrow G \cdot R^2 \cdot R^{-1} \mod N$
3: $X_1 \leftarrow \text{MontMul}(G, R^2 \mod N)$
4: $X_1 \leftarrow \text{SmallRed}(X_1)$
5: **for** $i = k-1$ **downto** 0 **do**
6: $\quad X_{1-e_i}, X_{e_i} \leftarrow \text{CombinedMontMul}(X_{e_i}, X_{1-e_i}, X_{e_i})$
7: // conversion $X_0 \leftarrow (G^e \cdot R) \cdot R^{-1} \mod N$
8: $X_0 \leftarrow \text{MontMul}(X_0, 1), X_0 \leftarrow \text{SmallRed}(X_0)$
9: **return** X_0

Using the complexities of Table V, one has:

$$\begin{aligned} \#\text{ADD} &= k(6n^2+9n+1) + (8n^2+8n), \\ \#\text{MUL} &= k(3n^2+4n+3) + (4n^2+4n+2). \end{aligned}$$

B. Right-to-left 2^t-ary exponentiation with CombinedMontMul

We first review the right-to-left m-ary method for modular exponentiation $G^e \mod N$. Let us assume that the exponent e is recoded as $e = (e_{k-1}, \ldots, e_0)_m$ where $e_i \in \{1, \ldots, m\}$ and $m = 2^t$ using the method of [1]. The right-to-left version of the m-ary exponentiation is based on the following expression of $G^e \mod N$:

$$\begin{aligned} G^e \mod N &= \textstyle\prod_{i=0,e_i=1}^{i=k-1} G^{m^i} \cdot \prod_{i=0,e_i=2}^{i=k-1} G^{2\cdot m^i} \cdots \\ &\quad \cdots \textstyle\prod_{i=0,e_i=m}^{i=k-1} G^{m\cdot m^i} \mod N \\ &= \textstyle\prod_{j=1}^{m} Y_j^j \mod N \end{aligned}$$

where $Y_j = \prod_{i=0,e_i=j}^{i=k-1} G^{m^i} \mod N$.

Consequently, $G^e \mod N$ is computed through a sequence $X_i \leftarrow G^{m^i} \mod N$ followed by $Y_{e_i} \leftarrow Y_{e_i} \cdot X_i$ for $i = 0, \ldots, k-1$. At the end, one gets Y_j for $j = 1, \ldots, m$. The final result $G^e = \prod_{j=1}^{m} Y_j^j$ is computed as follows with $2(m-1)$ multiplications:

$Z \leftarrow Y_m$
for $i = m-1$ **downto** 1 **do**
$\quad Y_i \leftarrow Y_i \cdot Y_{i+1} \mod N$
$\quad Z \leftarrow Z \cdot Y_i \mod N$

And at the end, one has $Z = G^e \mod N$.

When $m = 2^t$, we can take advantage of the CombinedMontMul algorithm to improve the right-to-left m-ary method. Indeed, the operations $Y_{e_i} \leftarrow Y_{e_i} \cdot X_i$ and $X_{i+1} \leftarrow X_i^{2^t}$ can be performed by first computing $Y_{e_i}, X_{i+1} \leftarrow \text{CombinedMontMul}(X_i, Y_{e_i}, X_i)$ and afterwards by computing $t-1$ Montgomery squarings $X_{i+1} \leftarrow \text{MontSqu}(X_{i+1})$ followed by $\text{SmallRed}(X_{i+1})$.

The final reconstruction $Z = \prod_{j=1}^{m} Y_j^j$ can also take advantage of the CombinedMontMul algorithm. Indeed, it consists of a sequence of two multiplications with a common operand: $Y_i \cdot Y_{i+1} \mod N$ and $Z \cdot Y_i \mod N$. This can also be performed using a sequence of $2^t - 2$ CombinedMontMuls.

The resulting improved right-to-left 2^t-ary exponentiation is shown in Algorithm 9. Its complexity is evaluated step by step in Table VI.

Algorithm 9 Right-to-left regular 2^t-ary exponentiation with CombinedMontMul

Require: $N < 2^{wn-2}$ the modulus, an integer $0 \le G < N$, an exponent $e = (e_{k-1}, \ldots, e_0)_{2^t}$ with $e_i \in \{1, \ldots, 2^t\}$, $R = 2^{w(n+1)}$ the Montgomery constant.
Ensure: $G^e \mod N$
1: // $X = G \cdot R \mod N$
2: $X \leftarrow \text{MontMul}(G, R^2 \mod N), X \leftarrow \text{SmallRed}(X)$
3: **for** $i = 1$ to 2^t **do**
4: $\quad Y_i \leftarrow R \mod N$
5: **for** $i = 0$ to $k-1$ **do**
6: $\quad Y_{e_i}, X \leftarrow \text{CombinedMontMul}(X, Y_{e_i}, X)$
7: $\quad$ **for** $j = 1$ **to** $t-1$ **do**
8: $\quad\quad X \leftarrow \text{MontSqu}(X), X \leftarrow \text{SmallRed}(X)$
9: // Final reconstruction
10: $Z \leftarrow Y_{2^t}$
11: $Y_{2^t-1} \leftarrow \text{MontMul}(Y_{2^t-1}, Y_{2^t})$
12: $Y_{2^t-1} \leftarrow \text{SmallRed}(Y_{2^t-1})$
13: **for** $i = 2^t - 1$ **downto** 2 **do**
14: $\quad Z, Y_{i-1} \leftarrow \text{CombinedMontMul}(Y_i, Z, Y_{i-1})$
15: $Z \leftarrow \text{MontMul}(Z, Y_1),\ Z \leftarrow \text{SmallRed}(Z)$
16: $Z \leftarrow \text{MontMul}(Z, 1), Z \leftarrow \text{SmallRed}(Z)$
17: **return** Z

Table VI
COMPLEXITY OF ALGORITHM 9

	Op.	# ADD	# MUL
Step 2	MM	$4n^2+2n-1$	$2n^2+n$
	SR	$2n+1$	$n+1$
k Step 6	CMM	$k(6n^2+9n+1)$	$k(3n^2+4n+3)$
$(t-1)k$	MS	$(t-1)k(3n^2+5n-1)$	$(t-1)k(\frac{3n^2}{2}+\frac{5n}{2}-1)$
Step 8	SR	$(t-1)k(2n+1)$	$(t-1)k(n+1)$
Step 11	MM	$4n^2+2n-1$	$2n^2+n$
Step 12	SR	$2n+1$	$n+1$
(2^t-2) Step 14	CMM	$(2^t-2)(6n^2+9n+1)$	$(2^t-2)(3n^2+4n+3)$
Step 15	MM	$4n^2+2n-1$	$2n^2+n$
	SR	$2n+1$	$n+1$
Step 16	MM	$4n^2+2n-1$	$2n^2+n$
	SR	$2n+1$	$n+1$
Total		$tk(3n^2+7n)$ $+k(3n^2+2n+1)$ $+2^t(6n^2+9n+1)$ $+4n^2-2n-2$	$tk(\frac{3n^2}{2}+\frac{7n}{2})$ $+k(\frac{3n^2}{2}+\frac{n}{2}+3)$ $+2^t(3n^2+4n+3)$ $+2n^2-2$

CMM=CombinedMontMul, MM=MontMul, MS=MontSqu, SR=SmallRed

C. Left-to-right 2^t-ary exponentiation using MultByComOp

In this subsection, we present an improved version of the left-to-right 2^t-ary exponentiation using the multiple multiplications by the same operand (Subsection III-B). The left-to-right exponentiation consists in first precomputing $G_i = G^i \mod N, i = 1, \ldots, 2^t$ and then in computing $X = G^e \mod N$ through the sequence of operations $X \leftarrow X^{2^t} \cdot G_{e_i}$ for $i = k-1, \ldots, 0$.

One can take advantage of the MultByComOp algorithm as follows:

- At the very beginning, one gets $G^{(0)}, \ldots, G^{(n-1)}$ with PrecomMultByComOp(G) in order to compute efficiently the $2^t - 1$ multiplications $G_i \leftarrow G \cdot G_{i-1} \mod N$.
- Then for each G_i, one gets $G_i^{(j)} = G_i \cdot 2^{-w(n-1-j)} \mod N$ for $j = 0, \ldots, n-1$ by computing PrecompMultByComOp(G_i).
- In the main loop, which sequentially computes $X \leftarrow X^{2^t} \cdot G_{e_i}$ for $i = k-1, \ldots, 0$, each multiplication by G_{e_i} is done as $X \leftarrow$ MultByComOp$(X, G_{e_i}^{(0)}, \ldots, G_{e_i}^{(n-1)})$.

The resulting improved modular exponentiation is given in Algorithm 10. The reader may notice that in the sequence of squarings for the computation of X^{2^t}, each Montgomery squaring or multiplication is followed by a SmallRed, in order to have the correct Montgomery factor $R = 2^{-w(n+1)}$.

Algorithm 10 Left-to-right regular 2^t-ary exponentiation with MultByComOp

Require: $N < 2^{wn-1}$ the modulus, an integer $0 \leq G < N$, an exponent $e = (e_{k-1}, \ldots, e_0)_2$ with $e_{k-1} \in \{1, \ldots, m\}$, $R = 2^{w(n+1)}$ the Montgomery constant.
Ensure: $G^e \mod N$

1: // $G_1 = G \cdot 2^{w(n+1)} \mod N$
2: $G_1 \leftarrow$ MontMul$(G, R^2 \mod N)$
3: $G_1 \leftarrow$ SmallRed(G_1)
4: $X \leftarrow R$
5: $G_1^{(0)}, \ldots, G_1^{(n-1)} \leftarrow$ PrecompMultByComOp(G_1)
6: **for** $i = 2$ **to** 2^t **do**
7: $\quad G_i \leftarrow$ MultByComOp$(G_{i-1}, G_1^{(0)}, \ldots, G_1^{(n-1)})$
8: $\quad G_i^{(0)}, \ldots, G_i^{(n-1)} \leftarrow$ PrecompMultByComOp(G_i)
9: **for** $i = k-1$ **downto** 0 **do**
10: $\quad$ **for** $j = 1$ **to** t **do**
11: $\quad\quad$ // $X \leftarrow X^2 \cdot 2^{-w(n+1)} \mod N$
12: $\quad\quad X \leftarrow$ MontSqu$(X), X \leftarrow$ SmallRed(X)
13: $\quad X \leftarrow$ MultByComOp$(X, G_{e_i}^{(0)}, \ldots, G_{e_i}^{(n-1)})$
14: // $X \leftarrow X \cdot 2^{-w(n+1)} \mod N$
15: $X \leftarrow$ MontMul$(X, 1), X \leftarrow$ SmallRed(X)
16: **return** X

We point out that this left-to-right version requires a large memory, i.e., $\cong n \times 2^t \times nw$ bits, to store the $n \times 2^t$ terms $G_i^{(j)}$.

Table VII
COMPLEXITY OF ALGORITHM 10

	Op.	# ADD	# MUL
Step 2	MM	$4n^2 + 2n - 1$	$2n^2 + n$
Step 3	SR	$2n + 1$	$n + 1$
Step 5	PMBCO	$2n^2 - n - 1$	$n^2 - 1$
$(2^t - 1)$ Step 7	MBCO	$(2^t - 1)(2n^2 + 5n + 1)$	$(2^t - 1)(n^2 + 2n + 2)$
$(2^t - 1)$ Step 8	PMBCO	$(2^t - 1)(2n^2 - n - 1)$	$(2^t - 1)(n^2 - 1)$
tk Step 12	MS SR	$tk(3n^2 + 5n - 1)$ $tk(2n + 1)$	$tk(\frac{3n^2}{2} + \frac{5n}{2} - 1)$ $tk(n + 1)$
k Step 13	MBCO	$k(2n^2 + 5n + 1)$	$k(n^2 + 2n + 2)$
Step 15	MM SR	$4n^2 + 2n - 1$ $2n + 1$	$2n^2 + n$ $n + 1$
Total		$tk(3n^2 + 7n)$ $+k(2n^2 + 5n + 1)$ $+2^t(4n^2 + 4n)$ $+6n^2 + 3n - 1$	$tk(\frac{3n^2}{2} + \frac{7n}{2})$ $+k(n^2 + 2n + 2)$ $+2^t(2n^2 + 2n + 1)$ $+3n^2 + 2n$

PMBCO=PrecompMulByComOp, MBCO=MulByComOp, MM=MontMul, MS=MontSqu, SR=SmallRed

V. COMPLEXITY AND IMPLEMENTATION

A. Complexity comparison

In Table VIII, we provide the complexities of the three considered approaches: Montgomery-ladder, right-to-left and left-to-right 2^t-ary exponentiations. For each exponentiation method, we first give the complexities without any optimization, i.e., using regular Montgomery multiplication and squaring. We then report the complexities of the proposed improved exponentiations, i.e., Algorithm 8, Algorithm 9 and Algorithm 10.

Table VIII
COMPLEXITIES OF MODULAR EXPONENTIATION ALGORITHMS

	ADD	MUL
Mont-ladder	$k(7n^2 + 7n - 2)$ $+8n^2 + 4n - 2$	$k(\frac{7n^2}{2} + \frac{7n}{2} - 1)$ $+4n^2 + 2n$
Mont-ladder with CMM	$k(6n^2 + 9n + 1)$ $+(8n^2 + 8n)$	$k(3n^2 + 4n + 3)$ $+(4n^2 + 4n + 2)$
Right-to-left	$tk(3n^2 + 5n - 1)$ $+k(4n^2 + 2n - 1)$ $+2^t(8n^2 + 4n - 2)$	$tk(\frac{3n^2}{2} + \frac{5n}{2} - 1)$ $+k(2n^2 + n)$ $+2^t(4n^2 + 2n)$
Right-to-left with CMM	$tk(3n^2 + 7n)$ $+k(3n^2 + 2n + 1)$ $+2^t(6n^2 + 9n + 1)$ $+4n^2 - 2n - 2$	$tk(\frac{3n^2}{2} + \frac{7n}{2})$ $+k(\frac{3n^2}{2} + \frac{n}{2} + 3)$ $+2^t(3n^2 + 4n + 3)$ $+2n^2 - 2$
Left-to-right	$tk(3n^2 + 5n - 1)$ $+k(4n^2 + 2n - 1)$ $+2^t(4n^2 + 2n - 1)$ $+4n^2 + 2n - 1$	$tk(\frac{3n^2}{2} + \frac{5n}{2} - 1)$ $+k(2n^2 + n)$ $+2^t(2n^2 + n)$ $+2n^2 + n$
Left-to-right with MBCO	$tk(3n^2 + 7n)$ $+k(2n^2 + 5n + 1)$ $+2^t(4n^2 + 4n)$ $+6n^2 + 3n - 1$	$tk(\frac{3n^2}{2} + \frac{7n}{2})$ $+k(n^2 + 2n + 2)$ $+2^t(2n^2 + 2n + 1)$ $+3n^2 + 2n$

We first consider the Montgomery-ladder: the improvement provided by the CombinedMontMul reduces the leading term of the ADD complexity from $7kn^2$ to $6kn^2$ and

the leading of the MUL complexity from $3.5kn^2$ to $3kn^2$. This represents an improvement of 14%.

For the right-to-left 2^t-ary exponentiation, the leading terms for both ADD and MUL complexities correspond to kt and are due to the sequence of squarings which are roughly the same for right-to-left and right-to-left with CMM. However, concerning the terms in k and 2^t, a reduction by $\cong 25\%$ is observed.

Finally, for the left-to-right methods, the leading terms in tk and in 2^t are roughly the same for left-to-right and left-to-right with MulByComOp. However, a reduction by $\cong 50\%$ for the term in k is observed.

As example, for a modulus N of 2048 bits, which is commonly used in RSA cryptosystems, we provide the explicit complexities of the modular exponentiations in Table IX. The improvement is around 13% for the Montgomery-ladder, 4% for the right-to-left exponentiation and 8% for the left-to-right exponentiation.

Table IX
COMPLEXITIES OF MODULAR EXPONENTIATION FOR 2048 BITS

	ML	ML CMM	R-to-L	R-to-L CMM	L-ro-R	L-to-R MBCO
#ADD/10^3	15143	13183	8595	8253	8466	7804
Improv.	12.9%		4%		7.9%	
#MUL/10^3	7506	6564	4296	4120	4232	3896
Improv.	12.6%		4.1%		7.9%	

ML=Montgomery-ladder, R-to-L= Right-to-left, L-to-R=Left-to-right

B. Software implementation results

The three considered exponentiation algorithms were coded in C, compiled with `gcc 4.8.2` and run on an Intel Core i7®-4770 CPU @ 3.4GHz. We used the low level functions performing word-multiplications and additions of the `GMP` library (`GMP 6.0.0`, `https://gmplib.org`) as building blocks of our codes. The timings were obtained by deactivating turbo-boost and hyperthreading options and by averaging the times of a few hundred executions with random inputs. The resulting timings are reported in Table X.

Table X
MODULAR EXPONENTIATION TIMINGS (10^3 CYCLES)

Algorithm	1024 bits		2048 bits		4096 bits	
	#CC/10^3	Imp. ratio	#CC/10^3	Imp. ratio	#CC/10^3	Imp. ratio
Mont-ladder	3068	9.0%	20643	9.1%	153443	14.6%
Mont-ladder with CMM	2793		18773		131011	
Right-to-left	1857	0.1%	11796	1.7%	87081	4.0%
Right-to-left with CMM	1855		11596		83599	
Left-to-right	1858	-1%	11734	3.1 %	83354	8.3 %
Left-to-right with MBCO	1877		11368		77745	

The reported timings show significant improvements for the Montgomery ladder for all considered sizes. For the right-to-left exponentiation, the improvement ratio is low as expected, but close to the theoretical ratio for 4096 bits. Finally, the left-to-right exponentiation with MBCO shows an improved timings only for 2048 and 4096 bits. Thus, the influence of the large memory space (256KB for 2048 bits and 1MB for 4096 bits) seems to be negligible in our tests, probably due to the fact that it is smaller than the L3 cache memory size.

VI. CONCLUSION

In this paper, we considered modular exponentiations for sizes corresponding to practical implementations of RSA. Multiplications and squarings modulo N are performed with the costly method of Montgomery [6] due to the random form of the modulus N. We showed that for multiplications which share a common operand, one can save some redundant computations. Indeed, the proposed approach reduces by 25% the computation of two products AB, AC. Extending this idea to a large number of multiplications by a common operand A, the complexity of each multiplication is improved by $\cong 50\%$. Based on these optimized modular multiplications, we improved modular exponentiations protected against simple power analysis and timing attacks. In particular we improved the Montgomery-ladder by $9-14\%$ compared to usual implementation involving word level Montgomery multiplication and squaring.

Acknowledgement. This work is partly supported by the PAVOIS project (ANR 12 BS02 002 01) and by the Thelxinoë Erasmus Mundus European project.

REFERENCES

[1] M. Joye and M. Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 334–349, 2009.

[2] M. Joye and S.-M. Yen. The Montgomery Powering Ladder. In *CHES 2002*, volume 2523 of *LNCS*, pages 291–302, 2002.

[3] P.C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113, 1996.

[4] P.C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397, 1999.

[5] A. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[6] P. Montgomery. Modular Multiplication Without Trial Division. *Math. Computation*, 44:519–521, 1985.

[7] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.